**Python Objects and Classes**

In the last tutorial, we learned about [Python OOP](https://www.programiz.com/python-programming/object-oriented-programming). We know that python also supports the concept of objects and classes.

An object is simply a collection of data (variables) and methods (functions). Similarly, a class is a blueprint for that object.

Before we learn about objects, let's first know about classes in Python.

## Python Classes

A class is considered as a blueprint of objects. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

## Define Python Class

We use the class keyword to create a class in Python. For example,

class ClassName:

 # class definition

Here, we have created a class named ClassName.

Let's see an example,

class Bike:

 name = ""

 gear = 0

Here,

* Bike - the name of the class
* name/gear - variables inside the class with default values "" and **0** respectively.

**Note**: The variables inside a class are called attributes.

## Python Objects

An object is called an instance of a class. For example, suppose Bike is a class then we can create objects like bike1, bike2, etc from the class.

Here's the syntax to create an object.

objectName = ClassName()

Let's see an example,

# create class

class Bike:

 name = ""

 gear = 0

# create objects of class

bike1 = Bike()

Here, bike1 is the object of the class. Now, we can use this object to access the class attributes.

## Access Class Attributes Using Objects

We use the . notation to access the attributes of a class. For example,

# modify the name attribute

bike1.name = "Mountain Bike"

# access the gear attribute

bike1.gear

Here, we have used bike1.name and bike1.gear to change and access the value of name and gear attribute respectively.

## Example 1: Python Class and Objects

# define a class

class Bike:

 name = ""

 gear = 0

# create object of class

bike1 = Bike()

# access attributes and assign new values

bike1.gear = 11

bike1.name = "Mountain Bike"

print(f"Name: {bike1.name}, Gears: {bike1.gear} ")

**Output**

Name: Mountain Bike, Gears: 11

In the above example, we have defined the class named Bike with two attributes: name and gear.

We have also created an object bike1 of the class Bike.

Finally, we have accessed and modified the attributes of an object using the . notation.

## Create Multiple Objects of Python Class

We can also create multiple objects from a single class. For example,

# define a class

class Employee:

 # define an attribute

 employee\_id = 0

# create two objects of the Employee class

employee1 = Employee()

employee2 = Employee()

# access attributes using employee1

employee1.employeeID = 1001

print(f"Employee ID: {employee1.employeeID}")

# access attributes using employee2

employee2.employeeID = 1002

print(f"Employee ID: {employee2.employeeID}")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Employee ID: 1001

Employee ID: 1002

In the above example, we have created two objects employee1 and employee2 of the Employee class.

## Python Methods

We can also define a function inside a Python class. A [Python Function](https://www.programiz.com/python-programming/function) defined inside a class is called a method.

Let's see an example,

# create a class

class Room:

 length = 0.0

 breadth = 0.0

 # method to calculate area

 def calculate\_area(self):

 print("Area of Room =", self.length \* self.breadth)

# create object of Room class

study\_room = Room()

# assign values to all the attributes

study\_room.length = 42.5

study\_room.breadth = 30.8

# access method inside class

study\_room.calculate\_area()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Area of Room = 1309.0

In the above example, we have created a class named Room with:

* **Attributes**: length and breadth
* **Method**: calculate\_area()

Here, we have created an object named study\_room from the Room class. We then used the object to assign values to attributes: length and breadth.

Notice that we have also used the object to call the method inside the class,

study\_room.calculate\_area()

Here, we have used the . notation to call the method. Finally, the statement inside the method is executed.

## Python Constructors

Earlier we assigned a default value to a class attribute,

class Bike:

 name = ""

...

# create object

bike1 = Bike()

However, we can also initialize values using the constructors. For example,

class Bike:

 # constructor function

 def \_\_init\_\_(self, name = ""):

 self.name = name

bike1 = Bike()

Here, \_\_init\_\_() is the constructor function that is called whenever a new object of that class is instantiated.

The constructor above initializes the value of the name attribute. We have used the self.name to refer to the name attribute of the bike1 object.

If we use a constructor to initialize values inside a class, we need to pass the corresponding value during the object creation of the class.

bike1 = Bike("Mountain Bike")

Here, "Mountain Bike" is passed to the name parameter of \_\_init\_\_().

# Python Inheritance

Like any other OOP languages, Python also supports the concept of class inheritance.

Inheritance allows us to create a new class from an existing class.

The new class that is created is known as **subclass** (child or derived class) and the existing class from which the child class is derived is known as **superclass** (parent or base class).

## Python Inheritance Syntax

Here's the syntax of the inheritance in Python,

# define a superclass

class super\_class:

 # attributes and method definition

# inheritance

class sub\_class(super\_class):

 # attributes and method of super\_class

 # attributes and method of sub\_class

Here, we are inheriting the sub\_class class from the super\_class class.

## Example 1: Python Inheritance

class Animal:

 # attribute and method of the parent class

 name = ""

 def eat(self):

 print("I can eat")

# inherit from Animal

class Dog(Animal):

 # new method in subclass

 def display(self):

 # access name attribute of superclass using self

 print("My name is ", self.name)

# create an object of the subclass

labrador = Dog()

# access superclass attribute and method

labrador.name = "Rohu"

labrador.eat()

# call subclass method

labrador.display()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I can eat

My name is Rohu

In the above example, we have derived a subclass Dog from a superclass Animal. Notice the statements,

labrador.name = "Rohu"

labrador.eat()

Here, we are using labrador (object of Dog) to access name and eat() of the Animal class. This is possible because the subclass inherits all attributes and methods of the superclass.

Also, we have accessed the name attribute inside the method of the Dog class using self.

## is-a relationship

In Python, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an **is-a** relationship between two classes. For example,

1. **Car** is a **Vehicle**
2. **Apple** is a **Fruit**
3. **Cat** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Apple** can inherit from **Fruit**, and so on.

### Example 2: Inheritance in Python

Let's take a look at another example of inheritance in Python,

A polygon is a closed figure with **3** or more sides. Say, we have a class called Polygon defined as follows,

class Polygon:

 def \_\_init\_\_(self, no\_of\_sides):

 self.n = no\_of\_sides

 self.sides = [0 for i in range(no\_of\_sides)]

 def inputSides(self):

 self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

 def dispSides(self):

 for i in range(self.n):

 print("Side",i+1,"is",self.sides[i])

This class has data attributes to store the number of sides n and magnitude of each side as a list called sides.

* The inputSides() method takes in the magnitude of each side
* The dispSides() method displays these side lengths

A triangle is a polygon with **3** sides. So, we can create a class called Triangle which **inherits** from Polygon. This makes all the attributes of Polygon class available to the Triangle class.

We don't need to define them again **(code reusability)**. Triangle can be defined as follows.

class Triangle(Polygon):

 def \_\_init\_\_(self):

 Polygon.\_\_init\_\_(self,3)

 def findArea(self):

 a, b, c = self.sides

 # calculate the semi-perimeter

 s = (a + b + c) / 2

 area = (s\*(s-a)\*(s-b)\*(s-c)) \*\* 0.5

 print('The area of the triangle is %0.2f' %area)

However, the Triangle class has a new method findArea() to find and print the area of the triangle.

Now let's see the complete working code of the example above including creating an object,

class Polygon:

 # Initializing the number of sides

 def \_\_init\_\_(self, no\_of\_sides):

 self.n = no\_of\_sides

 self.sides = [0 for i in range(no\_of\_sides)]

 def inputSides(self):

 self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

 # method to display the length of each side of the polygon

 def dispSides(self):

 for i in range(self.n):

 print("Side",i+1,"is",self.sides[i])

class Triangle(Polygon):

 # Initializing the number of sides of the triangle to 3 by

 # calling the \_\_init\_\_ method of the Polygon class

 def \_\_init\_\_(self):

 Polygon.\_\_init\_\_(self,3)

 def findArea(self):

 a, b, c = self.sides

 # calculate the semi-perimeter

 s = (a + b + c) / 2

 # Using Heron's formula to calculate the area of the triangle

 area = (s\*(s-a)\*(s-b)\*(s-c)) \*\* 0.5

 print('The area of the triangle is %0.2f' %area)

# Creating an instance of the Triangle class

t = Triangle()

# Prompting the user to enter the sides of the triangle

t.inputSides()

# Displaying the sides of the triangle

t.dispSides()

# Calculating and printing the area of the triangle

t.findArea()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Enter side 1 : 3

Enter side 2 : 5

Enter side 3 : 4

Side 1 is 3.0

Side 2 is 5.0

Side 3 is 4.0

The area of the triangle is 6.00

Here, we can see that even though we did not define methods like inputSides() or dispSides() for class Triangle separately, we were able to use them.

If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

## Method Overriding in Python Inheritance

In the previous example, we see the object of the subclass can access the method of the superclass.

**However, what if the same method is present in both the superclass and subclass?**

In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Python.

### Example: Method Overriding

class Animal:

 # attributes and method of the parent class

 name = ""

 def eat(self):

 print("I can eat")

# inherit from Animal

class Dog(Animal):

 # override eat() method

 def eat(self):

 print("I like to eat bones")

# create an object of the subclass

labrador = Dog()

# call the eat() method on the labrador object

labrador.eat()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I like to eat bones

In the above example, the same method eat() is present in both the Dog class and the Animal class.

Now, when we call the eat() method using the object of the Dog subclass, the method of the Dog class is called.

This is because the eat() method of the Dog subclass overrides the same method of the Animal superclass.

## The super() Method in Python Inheritance

Previously we saw that the same method in the subclass overrides the method in the superclass.

However, if we need to access the superclass method from the subclass, we use the super() method. For example,

class Animal:

 name = ""

 def eat(self):

 print("I can eat")

# inherit from Animal

class Dog(Animal):

 # override eat() method

 def eat(self):

 # call the eat() method of the superclass using super()

 super().eat()

 print("I like to eat bones")

# create an object of the subclass

labrador = Dog()

labrador.eat()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I can eat

I like to eat bones

In the above example, the eat() method of the Dog subclass overrides the same method of the Animal superclass.

Inside the Dog class, we have used

# call method of superclass

super().eat()

to call the eat() method of the Animal superclass from the Dog subclass.

So, when we call the eat() method using the labrador object

# call the eat() method

labrador.eat()

Both the overridden and the superclass version of the eat() method is executed.

## Uses of Inheritance

1. Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
2. Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
3. Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.

# Python Multiple Inheritance

A [class](https://www.programiz.com/python-programming/class) can be derived from more than one superclass in Python. This is called multiple [inheritance](https://www.programiz.com/python-programming/inheritance).

For example, A class Bat is derived from superclasses Mammal and WingedAnimal. It makes sense because bat is a mammal as well as a winged animal.

Multiple Inheritance

## Python Multiple Inheritance Syntax

class SuperClass1:

 # features of SuperClass1

class SuperClass2:

 # features of SuperClass2

class MultiDerived(SuperClass1, SuperClass2):

 # features of SuperClass1 + SuperClass2 + MultiDerived class

Here, the MultiDerived class is derived from SuperClass1 and SuperClass2 classes.

## Example: Python Multiple Inheritance

class Mammal:

 def mammal\_info(self):

 print("Mammals can give direct birth.")

class WingedAnimal:

 def winged\_animal\_info(self):

 print("Winged animals can flap.")

class Bat(Mammal, WingedAnimal):

 pass

# create an object of Bat class

b1 = Bat()

b1.mammal\_info()

b1.winged\_animal\_info()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Mammals can give direct birth.

Winged animals can flap.

In the above example, the Bat class is derived from two super classes: Mammal and WingedAnimal. Notice the statements,

b1 = Bat()

b1.mammal\_info()

b1.winged\_animal\_info()

Here, we are using b1 (object of Bat) to access mammal\_info() and winged\_animal\_info() methods of the Mammal and the WingedAnimal class respectively.

## Python Multilevel Inheritance

In Python, not only can we derive a class from the superclass but you can also derive a class from the derived class. This form of inheritance is known as **multilevel inheritance**.

Here's the syntax of the multilevel inheritance,

class SuperClass:

 # Super class code here

class DerivedClass1(SuperClass):

 # Derived class 1 code here

class DerivedClass2(DerivedClass1):

 # Derived class 2 code here

Here, the DerivedClass1 class is derived from the SuperClass class, and the DerivedClass2 class is derived from the DerivedClass1 class.

Multilevel Inheritance in Python

## Example: Python Multilevel Inheritance

class SuperClass:

 def super\_method(self):

 print("Super Class method called")

# define class that derive from SuperClass

class DerivedClass1(SuperClass):

 def derived1\_method(self):

 print("Derived class 1 method called")

# define class that derive from DerivedClass1

class DerivedClass2(DerivedClass1):

 def derived2\_method(self):

 print("Derived class 2 method called")

# create an object of DerivedClass2

d2 = DerivedClass2()

d2.super\_method() # Output: "Super Class method called"

d2.derived1\_method() # Output: "Derived class 1 method called"

d2.derived2\_method() # Output: "Derived class 2 method called"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Super Class method called

Derived class 1 method called

Derived class 2 method called

In the above example, DerivedClass2 is derived from DerivedClass1, which is derived from SuperClass.

It means that DerivedClass2 inherits all the attributes and methods of both DerivedClass1 and SuperClass.

Hence, we are using d2 (object of DerivedClass2) to call methods from SuperClass, DerivedClass1, and DerivedClass2.

# Python Operator Overloading

In Python, we can change the way [operators](https://www.programiz.com/python-programming/operators) work for user-defined types.

For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called **operator overloading**.

## Python Special Functions

Class functions that begin with double underscore \_\_ are called special functions in Python.

The special functions are defined by the Python interpreter and used to implement certain features or behaviors.

They are called **"double underscore"** functions because they have a double underscore prefix and suffix, such as \_\_init\_\_() or \_\_add\_\_().

Here are some of the special functions available in Python,

|  |  |
| --- | --- |
| Function | Description |
| \_\_init\_\_() | initialize the attributes of the object |
| \_\_str\_\_() | returns a string representation of the object |
| \_\_len\_\_() | returns the length of the object |
| \_\_add\_\_() | adds two objects |
| \_\_call\_\_() | call objects of the class like a normal function |

## Example: + Operator Overloading in Python

To overload the + operator, we will need to implement \_\_add\_\_() function in the class.

With great power comes great responsibility. We can do whatever we like inside this function. But it is more sensible to return the Point object of the coordinate sum.

Let's see an example,

class Point:

 def \_\_init\_\_(self, x=0, y=0):

 self.x = x

 self.y = y

 def \_\_str\_\_(self):

 return "({0},{1})".format(self.x, self.y)

 def \_\_add\_\_(self, other):

 x = self.x + other.x

 y = self.y + other.y

 return Point(x, y)

p1 = Point(1, 2)

p2 = Point(2, 3)

print(p1+p2)

# Output: (3,5)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, what actually happens is that, when we use p1 + p2, Python calls p1.\_\_add\_\_(p2) which in turn is Point.\_\_add\_\_(p1,p2). After this, the addition operation is carried out the way we specified.

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

|  |  |  |
| --- | --- | --- |
| Operator | Expression | Internally |
| Addition | p1 + p2 | p1.\_\_add\_\_(p2) |
| Subtraction | p1 - p2 | p1.\_\_sub\_\_(p2) |
| Multiplication | p1 \* p2 | p1.\_\_mul\_\_(p2) |
| Power | p1 \*\* p2 | p1.\_\_pow\_\_(p2) |
| Division | p1 / p2 | p1.\_\_truediv\_\_(p2) |
| Floor Division | p1 // p2 | p1.\_\_floordiv\_\_(p2) |
| Remainder (modulo) | p1 % p2 | p1.\_\_mod\_\_(p2) |
| Bitwise Left Shift | p1 << p2 | p1.\_\_lshift\_\_(p2) |
| Bitwise Right Shift | p1 >> p2 | p1.\_\_rshift\_\_(p2) |
| Bitwise AND | p1 & p2 | p1.\_\_and\_\_(p2) |
| Bitwise OR | p1 | p2 | p1.\_\_or\_\_(p2) |
| Bitwise XOR | p1 ^ p2 | p1.\_\_xor\_\_(p2) |
| Bitwise NOT | ~p1 | p1.\_\_invert\_\_() |

## Overloading Comparison Operators

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

Here's an example of how we can overload the < operator to compare two objects the Person class based on their age:

class Person:

 def \_\_init\_\_(self, name, age):

 self.name = name

 self.age = age

 # overload < operator

 def \_\_lt\_\_(self, other):

 return self.age < other.age

p1 = Person("Alice", 20)

p2 = Person("Bob", 30)

print(p1 < p2) # prints True

print(p2 < p1) # prints False

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

True

False

Here, \_\_lt\_\_() overloads the < operator to compare the age attribute of two objects.

The \_\_lt\_\_() method returns,

* True - if the first object's age is less than the second object's age
* False - if the first object's age is greater than the second object's age

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

|  |  |  |
| --- | --- | --- |
| Operator | Expression | Internally |
| Less than | p1 < p2 | p1.\_\_lt\_\_(p2) |
| Less than or equal to | p1 <= p2 | p1.\_\_le\_\_(p2) |
| Equal to | p1 == p2 | p1.\_\_eq\_\_(p2) |
| Not equal to | p1 != p2 | p1.\_\_ne\_\_(p2) |
| Greater than | p1 > p2 | p1.\_\_gt\_\_(p2) |
| Greater than or equal to | p1 >= p2 | p1.\_\_ge\_\_(p2) |

## Advantages of Operator Overloading

Here are some advantages of operator overloading,

* Improves code readability by allowing the use of familiar operators.
* Ensures that objects of a class behave consistently with built-in types and other user-defined types.
* Makes it simpler to write code, especially for complex data types.
* Allows for code reuse by implementing one operator method and using it for other operators.